Étape 2 : Positions dans un espace torique

Jamila Sam & Jean-Cédric Chappelier, 2018

Version : 1.1

But

Implémentation d'une classe de base représentant des positions dans un environnement torique à deux dimensions et d'une classe représentant des objets positionnables dans un tel environnement.

Description

Les entités que vous serez amenés à simuler dans ce projet (fourmis, fourmilières, sources de nourriture etc.) vont évoluer dans un espace rectangulaire, à deux dimensions, de hauteur et largeur données. Pour les situer dans cet espace, il est nécessaire de leur associer une « *position* ». C'est cette notion qu'il s'agit de modéliser à cette étape.

Pour faire en sorte qu'un animal simulé se déplace, il faudra modifier sa position au cours du temps. La question se pose alors de comment gérer la situation où l'animal, suite à un déplacement, arrive aux confins de l'espace rectangulaire dans lequel la simulation se déroule. Pour répondre à cette problématique, nous faisons le choix de considérer l'environnement simulé comme « *torique* », c.-à-.d que l'on revient à gauche si l'on sort à droite, que l'on revient en haut si l'on sort en bas, et réciproquement (voir fig. 1 & 2). L'idée est que les animaux simulés vont évoluer dans un espace périodique, comme s'ils étaient sur un pneu (ou un donut !).

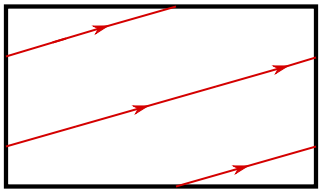

Figure 1: un monde torique.

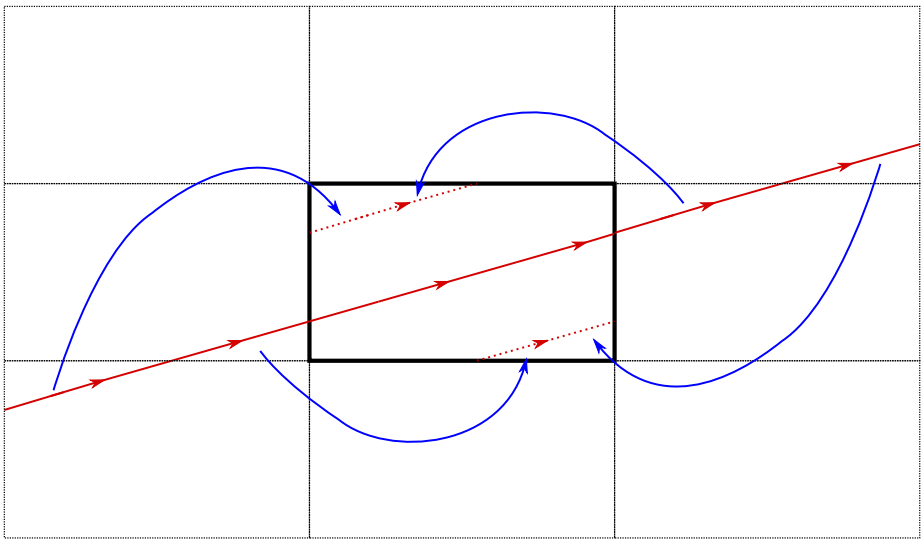

Figure 2: un monde torique « déplié » dans le monde usuel.

La première classe que vous avez à écrire représente donc la notion de position dans un environnement torique à deux dimensions, lequel sera représenté comme un rectangle, de hauteur et largeur données (voir fig. 1). Cette classe utilitaire sera omniprésente dans le projet.

Il vous est également demandé dans cette étape d'écrire la classe `Positionable` qui représente une entité ayant une position (dans notre monde torique). Toutes les entités de ce projet hériteront de cette classe.

Implémentation

Classe `ToricPosition`

La classe `ToricPosition` qu'il vous est demandé d'écrire permet de représenter une position dans un monde torique tel que décrit ci-desus. Cette classe devra être écrite dans le package `ch.epfl.moocprog`. Elle est caractérisée par une paire de coordonnées (une coordonnée en x et une en y) que vous représenterez par un objet de type `Vec2d`, classe qui vous est fournie. Une fois initialisé, ce `Vec2d` ne devra pas pouvoir être modifié.

De plus, il n'y a pas vraiment de sens à ce que la classe `ToricPosition` puisse être dérivée ; elle devra donc être `final`.

Vous doterez la classe `ToricPosition` des constructeurs et méthodes suivants :

1. une méthode `clampedPosition` (statique et privée) qui prend en paramètre deux `double` (`x` et `y`) et retourne un `Vec2d` « projeté » dans le monde torique (voir fig. 3) selon l'algorithme suivant :

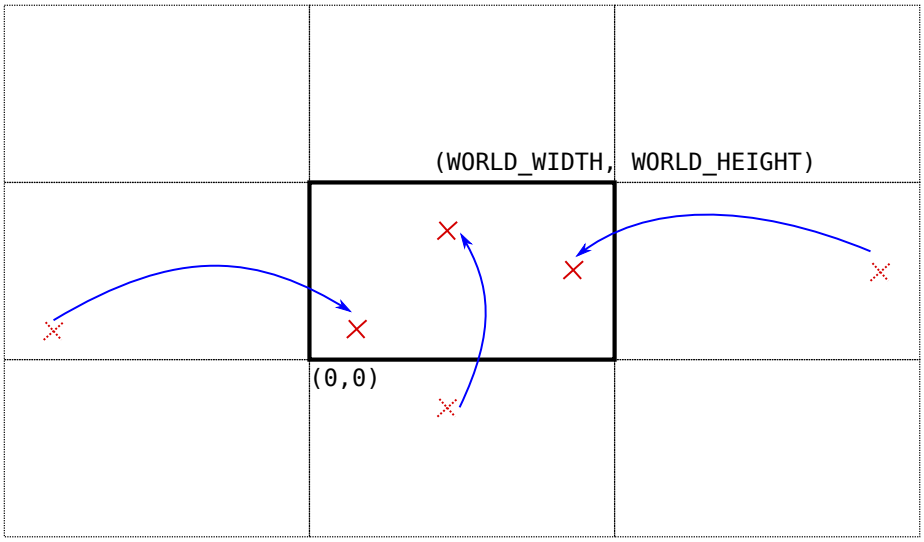

Figure 3: projection de coordonnées usuelles dans le monde torique.

tant que la valeur de la coordonnée `x` est inférieure strictement à zéro, l'augmenter de `WORLD_WIDTH`
et tant qu'elle est supérieure ou égale à `WORLD_WIDTH`, la décrémenter de cette même valeur ;
on procéde de façon analogue pour la coordonnée `y`, avec `WORLD_HEIGHT` au lieu de `WORLD_WIDTH` ;

vous pouvez accéder à la valeur de `WORLD_WIDTH` par l'appel à

```
getConfig().getInt(WORLD_WIDTH)
```

et de façon similaire pour `WORLD_HEIGHT` ;

Note : ces constantes sont stockées dans un fichier de configuration fourni comme décrit dans le document « *Complément : paramètres de simulation* » de la semaine 1 ; ainsi, elles peuvent être facilement modifiées sans devoir recompiler tout le projet ; n'oubliez pas les directives d'importation statiques que ce document vous décrit :

```
import static ch.epfl.moocprog.app.Context.getConfig;
import static ch.epfl.moocprog.config.Config.*;
```

2. un constructeur initialisant l'attribut de type `Vec2d` à partir de deux valeurs `x` et `y` fournies en paramètres ;
ce constructeur devra transformer les coordonnées passées en arguments

pour qu'elles correspondent à une position valide dans un environnement torique (c.-à-d. les projeter à l'aide de la méthode `clampedPosition`) ;

3. un constructeur initialisant l'attribut de type `Vec2d` à partir d'un `Vec2d` passé en paramètre, en appliquant toujours l'algorithme de « projection » ; notez la présence des méthodes `getX()` et `getY()` pour les `Vec2d` ;

4. un constructeur par défaut initialisant les deux coordonnées du `Vec2d` interne à 0.0 ;

5. une méthode

   ```
   ToricPosition add(ToricPosition that)
   ```

   qui retourne une nouvelle `ToricPosition` correspondant à l'addition des positions de l'instance courante (`this`) et de `that` ;
   utilisez pour cela la méthode `add()` de `Vec2d` ; le résultat doit être dans le monde torique (c.-à-d. « projeté » dans celui-ci) ;

6. une méthode

   ```
   ToricPosition add(Vec2d vec)
   ```

   qui est similaire à la méthode précédente ;

7. une méthode

   ```
   Vec2d toVec2d()
   ```

   qui retourne l'objet interne `Vec2d` ;

8. une méthode

   ```
   Vec2d toricVector(ToricPosition that)
   ```

   calculant le plus petit vecteur allant de l'instance courante (`this`) à `that` dans le monde torique ; en raison de l'aspect torique du monde, ce calcul est un peu subtil : il y a en effet plusieurs choix possibles pour calculer à quel vecteur les deux extrémités correspondent dans le monde torique (voir fig. 4) ;

   il y a en effet 9 trajets possibles vers `that` dans le monde torique, qui correspondent aux 9 cas suivants dans le monde habituel (voir fig. 4) :

   - `that` lui même ;
   - `that` augmenté ou décrémenté du vecteur `(0 , WORLD_HEIGHT)` ;
   - `that` augmenté ou décrémenté du vecteur `(WORLD_WIDTH, 0)` ;
   - `that` augmenté de `(+/- WORLD_WITH, +/- WORLD_HEIGHT)`, dans chacune des 4 combinaisons possibles ;

   l'algorithme appliqué sera alors le suivant : on calculera les 9 candidats possibles à l'aide de la méthode `add()` et on retient celui de plus faible distance à `this` ; aidez vous de la méthode `distance()` de la classe `Vec2d` pour cela ;

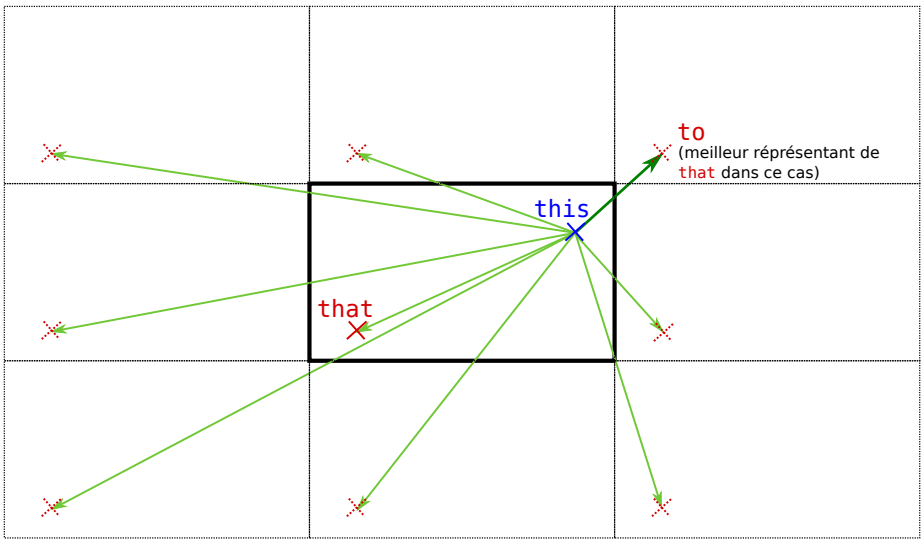

Figure 4: plus court chemin dans le monde torique.

9. une méthode

   `double toricDistance(ToricPosition that)`

   qui calcule la plus courte distance (dans le monde torique) entre `this` et `that` ; il suffit pour cela d'utiliser la méthode `toricVector()` et d'utiliser la méthode `length()` de la classe `Vec2d` pour en calculer la longueur ;

Pour finir, dotez la classe `ToricPosition` d'une méthode `toString()` qui retourne simplement le `toString()` de son `Vec2d`.

Classe `Positionable`

Créez la classe `Positionable` dans le package `ch.epfl.moocprog`. Cette classe représente une entité (quelconque, abstraction) ayant une position dans notre monde torique. Elle est donc caractérisée par sa position, de type `ToricPosition`.

Cette classe `Positionable` devra être dotée des méthodes suivantes :

- un constructeur par défaut qui initialise la position à (0.0, 0.0) ;
- un constructeur initialisant la position à la `ToricPosition` passée en paramètre ;
- une méthode publique `getPosition()` qui retourne la position ;

- une méthode `setPosition(ToricPosition position)` qui modifie la position ; cette méthode doit être protégée : changer la position d'une entité ne doit pas pouvoir être fait de l'extérieur du package. Elle sera également finale pour en éviter toute redéfinition dans les sous-classes.
- une méthode `toString()` qui retourne simplement le `toString()` de l'attribut modélisant la position.

Tests et soumission

Tests locaux

Dans `ch/epfl/moocprog/tests`, nous vous fournissons un fichier `Main.java` vous proposant quelques exemples de tests élémentaires. Une fois les classes complétez comme indiqué ci-dessus, vous devriez obtenir le résultat suivant avec ce fichier :

```
Some tests for ToricPosition
Default toric position : Position : 0.0, 0.0
tp2 : Position : 1.2, 2.3
tp3 : Position : 4.5, 6.7
tp4 (tp2 + tp3) : Position : 5.7, 9.0
Toric vector between tp2 and tp3 : Position : 3.3, 4.4
World dimension (clamped) : Position : 0.0, 0.0
Half world dimension : Position : 500.0, 350.0
tp3 + 2 * half world dimension = Position : 5.7, 9.0
Length of vector (3, 4) : 5.0

Some tests for Positionable
Default position : Position : 0.0, 0.0
Initialized at tp4 : Position : 5.7, 9.0
```

Vous pouvez modifier ce fichier comme vous le voulez. Nous vous encourageons d'ailleurs à le faire et à y ajouter tous **vos** tests, c.-à-d. tous les tests qui vous semblent pertinents et qui vous permettent de vérifier le bon fonctionnement de votre code.

Par exemple, si vous soumettez votre solution au correcteur et que vous n'obtenez pas tous les points, essayez d'ajouter un test similaire dans ce fichier test, puis corrigez votre code jusqu'à ce que ce test passe pour vous localement. Vous pourrez alors resoumettre votre code au correcteur automatique. Nous vous encourageons à garder comme cela le plus de tests possibles.

Plus tard dans le projet (semaine prochaine, étape 4) vous pourrez aussi utiliser une interface graphique pour voir ce qui se passe et avoir une meilleure intuition de ce qui, peut être, ne va pas. Mais gardez aussi toujours comme moyen

complémentaire ce fichier tests ci (`tests/Main.java`) pour faire des tests plus spécifiques, plus ponctuels. C'est la combinaison de ces **deux** outils (interfaces graphique d'une part et tests unitaires d'autre part) qui vous permettra de développer de façon efficace.

Soumission

Pour soumettre votre devoir au correcteur automatique, il faut créer un fichier ZIP contenant vos fichiers sources, **depuis la racine `ch`** (ce dernier point est important). Pour ce devoir ci, votre fichier ZIP doit donc contenir exactement les deux fichiers :

```
ch/epfl/moocprog/Positionable.java
ch/epfl/moocprog/ToricPosition.java
```

Note : si c'est plus pratique pour vous, vous *pouvez* aussi faire un zip qui contient tout le sous-répertoire `ch/epfl/moocprog`, y compris, donc, les fichiers que nous vous avons fournis. Ces fichiers seront simplement ignorés par le correcteur automatique (et remplacés par les nôtres). L'énoncé « *Procédure de soumission* » de la semaine 1 vous indique comment procéder.